Error Reporting Logic

Ciera Jaspan
Carnegie Mellon University
Pittsburgh, PA USA
ciera@cmu.edu

Trisha Quan
Carnegie Mellon University
Pittsburgh, PA USA
tkq@andrew.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, PA USA
aldrich@cs.cmu.edu

***Abstract*—When a system fails to meet its specification, it can be difficult to find the source of the error and determine how to fix it. In this paper, we introduce error reporting logic (ERL), an algorithm and tool that produces succinct explanations for why a target system violates a specification expressed in first order predicate logic. ERL analyzes the specification to determine which parts contributed to the failure, and it displays an error message specific to those parts. Additionally, ERL uses a heuristic to determine which object in the target system is responsible for the error. Results from a small user study suggest that the combination of a more focused error message and a responsible object for the error helps users to find the failure in the system more effectively. The study also yielded insights into how the users find and fix errors that may guide future research.**

I. Introduction

Many specification languages are based upon first-order predicate logic. This is a very natural route to take for specifications; it provides a concise, expressive, and well-understood way for describing system-level details. Examples of such specification languages in recent literature include Acme, SCL, and Alloy [1]–[3]. In each of these languages, FOPL-based specifications constrain a system, and a tool produces errors when there is an inconsistency between the specifications and the system. The error messages produced by these systems generally fall into three categories:

- *Specification identifier.* Under this mechanism, the tool produces an error message that states which specification failed. The user must read the specification and manually analyze the system to determine which part of the system broke the specification.
- *Human generated message.* This mechanism attempts to provide the user with an intuitive understanding of the specification. The specification writer makes a generic summary about what the specification is checking, and this is used as the error message. The user can then use this message as a guide to understand the general problem.
- *Hybrid systems.* Some tools also hybridize the two mechanisms; they will use a human generated error message if it exists, but they will fall back on a specification identifier.

These mechanisms work very well for specifications that are short and have an obvious point of failure. However, they do not work well for complex specifications, such as the Acme specification shown in Figure 1. By Acme standards, this is a medium sized specification. It has 3 levels of quantification, a very small inner predicate, and it only calls pre-defined atomic predicates.

```
forall con : ORMConnector in self.CONNECTORS |
 forall comp : Component in self.COMPONENTS |
  forall p : Port in comp.ports |
   (attached(con.caller, p) ->
    declaresType(comp, Data) and
    declaresType(p, DataPort))
```

Fig. 1. Sample Acme Specification

If the user must read the specification itself, they can quickly become lost in the details of the specification. There is no way to tell which sub-predicates in the specification failed, so the user must check each one. The user also doesn't know which elements in the system caused this failure and so must check all elements with matching types.

Even if the specification writer provided an error message, this would not necessarily help a user. An error message would tell us the purpose of the specification, and this might help the user look for bad patterns of behavior in the system. However, it still does not describe which predicate failed or which object in the system caused the failure.

In Figure 1, the user would have to check the entire system for conformance to the specification. What we would prefer is an error message that says:

myPort must declare the type DataPort since myConn.caller is attached to myPort

Error reporting logic (ERL) provides an automated way for creating error messages such as the one above. ERL presents each failing point as a unique error. To do this, it singles out only the failing predicates and assigns responsibility of the error to a specific object in the system.

In this paper, we will provide four contributions related to error messages from FOPL-based specifications:

- We present a user study that provides several insights into how users examine errors to find the root cause of the problem and how users attempt to fix the error. Primarily, we found that users see an error message as a single task which they must resolve, they only use keywords to find the problem rather than reading anything in depth, and they frequently rely on pattern recognition to find and fix errors. (Section II)
- We present ERL, a system for automatically generating error messages from a specification based on first-order

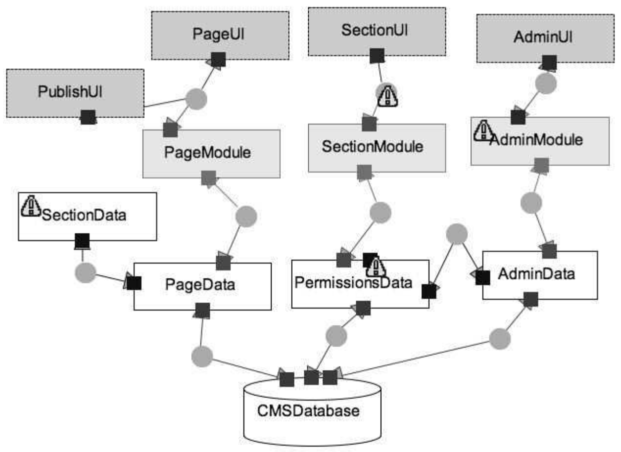

Fig. 2. The Web System in AcmeStudio

predicate logic. Section III will show how the ERL handles each of the specifications from the user study, and Section VI shows how the implementation of ERL performs with MDS, the most complex architectural specification built with Acme.

- We have implemented ERL as a reusable component and have integrated it within AcmeStudio [4]. The integration was relatively straightforward and required only a small amount of work to change the error messages. Section IV provides implementation and integration details.
- During the user study, the same participants also used ERL. In section V we describe how the users reacted to the new error messages. Three of the four participants benefited from the ERL error messages. The remaining participant did not benefit, but was not hindered, by the error messages.

Throughout this paper, we will use the Acme specification language (and AcmeStudio, the graphical interface and checker for Acme) as our example system. AcmeStudio allows developers to view a graphical representation of an architecture. While the developers can access and edit the Acme code behind the graphical view, it is typically not used. AcmeStudio displays the architecture using component-connector diagrams which can be edited entirely through a user interface. A sample diagram for an architecture is shown in Figure 2.

If an architecture fails to meet a specification, a red error triangle appears at the place where the specification was defined, as shown in Figure 2. Notice that this is not necessarily the component which is causing the failure. If the specification was defined at the system level, rather than the component level, then no error triangle appears.

In Acme, a software architect can choose to associate a handwritten error message to each specification. If the specification fails, for any reason, AcmeStudio displays the error message and a link to the specification code, in addition to the graphical indicator of the error. If the architect did not provide a default error message, then AcmeStudio displays only the link to the specification code. Since the architect can only write one message for the entire specification, the error message is typically about the general purpose of the specification.

TABLE I
PARTICIPANTS

ID	Configuration 1	Configuration 2
A	Web + ERL	Build
B	Web	Build + ERL
C	Build + ERL	Web
D	Build	Web + ERL

II. How users find specification errors

We ran a small user study of AcmeStudio to determine how users fix errors, with and without ERL loaded.[1] The four participants were Masters students and had used AcmeStudio to complete a couple of class assignments. The participants were told that the study was about usability in AcmeStudio and how developers find and fix errors; they were not told that the error messages were changed until after the main part of the study.

We provided the participants with two sets of Acme specifications, and we created an architecture for each which broke several of the specifications. The participants were asked to fix all the errors in the architectures. We asked participants to talk aloud while they worked, and we recorded each session with voice recordings, screen capture, and our observational notes.

Each participant used both sets of specifications, and each participant used AcmeStudio with and without ERL loaded. Participants were each assigned a different configuration in a different order, as shown in Table I.

The seeded errors were approximately equivalent in both systems. We created five categories of specifications, as detailed in Table II. Each system contained a broken specification from each category, and they were approximately the same level of difficulty to find and fix.

While Acme supports hand written error messages, they are infrequently used in practice, and we did not include them in the user study. The participants either received a message from ERL, or they received the name of the specification which broke. In both systems, the specification source was easily available by double-clicking on the error. The specifications were written in a style familiar to Acme users, and we only used atomic predicates of Acme which our users were already familiar with, found below.

- *binary relations* such as $<$, $>$, and $==$.
- *size(l)* to get the size of a list l.
- *attached(o1, o2)* to test if $o1$ is directly attached to $o2$.
- *declaresType(o, t)* to test if o declares the type t.

A. Results from the control

We will start by looking at how the participants fixed errors in the control system. This provided several insights into prob-

[1]The participants used an earlier version of ERL than is presented in Section III, but it had the same practical effect in the study.

TABLE II
BROKEN SPECIFICATIONS

Type of error	How broken in the given architecture	Example specification from the user study
Simple contains atomic predicates and at most 1 universal quantifier	The atomic fails once	`rule atLeastOneAttachedRole = size(self.ATTACHEDROLES) >= 1;`
Conjunction contains at least 1 conjunction of atomics and at most 2 quantifers	Both parts of the conjunction fail in one instance	`rule hasInAndOut = (exists p:Port in self.PORTS | declaresType(p,ORMPort)) and (exists p:Port in self.PORTS | declaresType(p,dataProvider));`
Quantification contains at least 2 quantifiers	The specification fails for two instances	`rule ResultsOnly = forall comp:DeployResults in self.COMPONENTS | forall p:inputPort in comp.PORTS | declaresType(p, resultsPort);`
Disjunction contains at least 1 disjunction and at most 1 quantifer	Both parts of the disjunction fail in one instance	`rule usingXMLRoles = forall r:Role in self.ROLES | declaresType(r, XMLReceiverRole) or declaresType(r, XMLProviderRole);`
Other may contain any other predicate and one universal quantifiers[2]	The predicate under test fails once.	`rule compilingIsOutput = forall p:Port in self.PORTS | declaresType(p, compilePort) -> declaresType(p, outputPort);`

lems users have with the existing error reporting mechanisms, including problems which ERL solves and problems for future work. In this section, we will look at trends we saw when participants from our user study used the original version of AcmeStudio. Later, in Section V, we will see how ERL helped several users find the root cause of the error.

With each of the errors, all of the participants used the graphical cues as a starting point. Of the five specifications, only four had graphical cues; one rule was defined at the system level and therefore had no graphical cue. Participant B investigated this error early on and decided to come back to it later because he could not easily find the location of the error.

The next step participants took was to read the specification source. When participants did this, they typically did not fully read the specification, but rather scanned it for keywords such as type names. When they flipped back to the architecture, they looked for a place where there were elements with those types all near each other. They would then investigate this area of the architecture thoroughly to determine whether something was obviously wrong, such as an unattached connector. The participants fully read the specification only when they could not find the problem through other means.

Participant D did attempt to fully read the specification to find an error, but soon ran into difficulties. The participant was working on the *conjunction* error from Table II and started by quickly reading through the specification. The participant noted that there were two parts to this specification and said

> "Does it tell me which side is failing?... Nope, no help."

[2]Build had a failing implication, and Web had a failing existential

The participant then spent several minutes trying to understand the specification and reviewing the architecture around the error location. This participant became more frustrated at this point, asking

> "But which part of the error is failing? It would be nice to see which part is failing so I don't have to parse it."

The final major technique participants used to find the error was to recognize and mimic "good" patterns. They would start noticing the patterns of how architectural elements were laid out and then check other elements to see if they conformed to the same pattern. In some cases, participants would believe they had found a problem that didn't actually exist, or they would find a problem that was different from the one they believed they were after. Upon finding any inconsistent patterns, the participant would attempt to make them identical.

This worked best if the participant understood the cause of the error before looking at the good example, or at least figured out the problem while they were looking at the example. If the participant did not understand the cause of the error, they could accidentally believe the correct example was actually the incorrect one. This problem occurred with the *quantification* errors from Table 2 since the participants could not tell which elements cause the failure.

Participant C used this technique quite frequently during the control part of the study. This participant made four comments about this during the study, usually comparing himself to a monkey:

> "Doing like a monkey, trying to match patterns..."

In several cases, Participant C found the inconsistency and fixed the error without ever knowing what the problem was.

B. Expectations for ERL

Based on the information from our user study, we believe that error reporting systems should:

- Direct users to the likely cause of the error, rather than the location where the specification is defined
- Assist users by including relevant keywords and excluding irrelevant ones
- Focus users on the part of the error they need to fix
- Provide users with examples that correctly pass the specifications

As we will see in the next section, ERL achieves three of the four objectives above. We leave the last task, providing users with examples, for future work, as it is not clear whether this will help users or possibly misdirect their attention.

III. Error Reporting Logic

In this section, we will see how ERL breaks down specifications to include only the relevant information about an error. To facilitate this explanation, we will use the error messages ERL produced during the study.

The intuition behind all of these rules is that ERL will produce an error for each "fix task" that the user must complete in order to make the specification succeed. Therefore, a user may have to complete several fix tasks before a specification passes, but each task can be thought of as logically independent from the other tasks. By doing this, ERL focuses the user on only a small portion of the overall failure and provides keywords relevant for only that portion.

ERL also associates each error message with a single object which is responsible for the error. This allows a system to add features such as jumping to a line number of code, or, in the case of AcmeStudio, indicating an architectural element as being at-fault. The choice of the *responsible object* is a heuristic because there is not enough information to tell which object is truly at-fault. For example, consider the case where the predicate $A(x) \implies B(y)$ fails. It is not clear whether $B(y)$ should be true, thus making y responsible, or whether $A(x)$ should be false, thereby causing x to be responsible for the error. Even in a single atomic predicate, the responsible object is ambiguous if there are multiple variables, such as the case in $equals(x, y)$. Therefore, ERL uses a heuristic based on the structure of the specification to determine which object is likely to be responsible. Regardless, ERL does guarantee that the responsible object was used in the failing predicate, even if it is not semantically the root cause.

The judgments for ERL are in the form

$$M, \Gamma \vdash p \hookrightarrow S$$

which is read as "Given the oracle M and context Γ, the predicate p produces the set of errors S".

Γ is a context that maps a unique variable name to a host-specific object. By *host*, we are referring to any FOPL-based specification system that uses ERL, such as Acme.

The *oracle* M is provided by the host specification system. The oracle provides answers to queries about *atomic predicates*, that is, a predicate which has some host-specific semantics. Our concept of an oracle is based on the concept of the oracle used in testing and [5]. In ERL, the oracle can be queried for the following:

- $\mathsf{evaluate}(\Gamma, a)$ evaluates whether the atomic predicate a is true, given the context provided in Γ.
- $\mathsf{items}(\Gamma, e)$ retrieves a list of objects for a quantifier, given some host-specific expression e.
- $\mathsf{freevars}(p)$ retrieves the free variables in p.
- $\mathsf{text}(\Gamma, p, isNegative, isDeontic)$ gets the message for the predicate p, given the context Γ. When p is an atomic predicate, this message is host-specific. We list a sampling of messages defined by the Acme oracle in Figure 3. If $isNegative$ is true, we must negate the message. If $isDeontic$ is true, the oracle produces a message in deontic mode ("a must be equal to b"), while if it is false, the message is stated as a fact ("a is equal to b").

The set S is a set of tuples $(r, \mathbf{x}, m)$ where m is the error message, r is the responsible object, the host-specific object that ERL will blame the error on, and $\mathbf{x}$ is the set of variables used to create m that are still unbound. Notice that for a single specification, the algorithm can produce multiple error messages, and each error message has its own responsible object. It is possible for the responsible object to have no value, represented in our rules as $\bullet$. In this situation, the host specification system may use its default assignment.[3]

The predicate p may be any first order logic predicate. ERL currently works for conjunction, disjunction, implication, negation, universal quantification, and existential quantification. Other first-order connectives, such as exclusive disjunction or unique quantification, can be added to ERL, but higher order predicates are not supported. Predicates also include any atomic predicates that are defined by the host specification system. Atomic predicates may be nested if the host system allows it, but ERL treats the entire predicate as an atomic predicate and will not descend into it.

In the remainder of the section, we will look at the rules for each logical connective. We will refer to the example specifications from Table II and also provide the error message and responsible object which ERL produces.

A. Simple Specifications

For the *simple* error shown in Table II, ERL produces the error message:

> *The size of interData1.AttachedRoles must be greater than or equal to 1.*
> (Responsible object: interData1)

The ERL rules for atomic predicates are:

$$\frac{M.\mathsf{evaluate}(\Gamma, a) = \mathtt{true}}{M, \Gamma \vdash a \hookrightarrow \emptyset}\,(A\text{-}\mathrm{TRUE})$$

[3]Acme assigns the error to the object which defined the specification; this is the `self` object in the example specifications.

$$\begin{aligned}
M.\mathsf{text}(\Gamma, declaresType(a,b), \mathtt{false}, \mathtt{true}) &= \Gamma(a) + \text{“ must declare the type”} + \Gamma(b)\\
M.\mathsf{text}(\Gamma, attached(a,b), \mathtt{false}, \mathtt{true}) &= \Gamma(a) + \text{“ must be attached to”} + \Gamma(b)\\
M.\mathsf{text}(\Gamma, equals(a,b), \mathtt{false}, \mathtt{true}) &= \Gamma(a) + \text{“ must be equal to”} + \Gamma(b)\\
M.\mathsf{text}(\Gamma, equals(a,b), \mathtt{true}, \mathtt{true}) &= \Gamma(a) + \text{“ must not be equal to”} + \Gamma(b)\\
M.\mathsf{text}(\Gamma, equals(a,b), \mathtt{false}, \mathtt{false}) &= \Gamma(a) + \text{“ is equal to”} + \Gamma(b)\\
M.\mathsf{text}(\Gamma, equals(a,b), \mathtt{true}, \mathtt{false}) &= \Gamma(a) + \text{“ is not equal to”} + \Gamma(b)
\end{aligned}$$

Fig. 3. Sampling of atomic messages for Acme

$$\frac{M.\mathsf{evaluate}(\Gamma, a) = \mathtt{false}}{\begin{array}{c}M, \Gamma \vdash a \hookrightarrow\\ \{(\bullet, M.\mathsf{freevars}(a), M.\mathsf{text}(\Gamma, a, \mathtt{false}, \mathtt{true}))\}\end{array}}\,(A{-}\mathtt{FALSE})$$

As the specification has only an atomic predicate, this was produced by directly querying the oracle for the truth of this statement and the error message. The message is stored in the error set. At the atomic level, we do not yet know which object will be responsible for this failure, so this is left as • for now. Section III-D will show how this is filled in.

B. Splitting errors

Our goal in ERL is to make each error represent a correction that the user must make in order to meet the specification. To do this, we will split errors upon evaluating a conjunction so that each side will produce a separate error. In ERL, the *conjunction* error from Table II produces two distinct errors:

> *There must exist a p in SectionData.Ports such that p declares the type dataProvider.*
> (Responsible object: SectionData)
> *There must exist a p in SectionData.Ports such that p declares the type ORMPort.*
> (Responsible object: SectionData)

The ERL rule that produces these errors, shown below, simply evaluates each side independently and unions the sets of errors together. Because of this split, the errors from the two sides may even have two different responsible objects.

$$\frac{M, \Gamma \vdash p_a \hookrightarrow S_a \qquad M, \Gamma \vdash p_b \hookrightarrow S_b}{M, \Gamma \vdash p_a \wedge p_b \hookrightarrow S_a \cup S_b}\,(\wedge)$$

C. Joining errors

When the user attempts to fix the *disjunction* error in Table II, they are working on a single task. Therefore, ERL shows a single error message. While the message can be lengthy, it contains all the keywords which a user might need to fix the error.

> *DataModelReceiver0 must declare the type XMLReceiverRole or DataModelReceiver0 must declare the type XMLProviderRole*
> (Responsible object: DataModelReceiver0)

This message was created by the ERL rule for joining messages on a disjunction failure, described below by rule ($\vee - \mathtt{false}$). Notice that if we have already split the error on both sides, we must rejoin all the splits into a single error message. This does mean that the error messages are much longer, but they are also specific to the task at hand. As an alternative to joining, if the specification system has hierarchical error reporting, ERL could create sub-errors and instruct the user to fix one sub error.

$$\frac{M, \Gamma \vdash p_a \hookrightarrow \emptyset \qquad M, \Gamma \vdash p_b \hookrightarrow S}{M, \Gamma \vdash p_a \vee p_b \hookrightarrow \emptyset}\,(\vee{-}\mathtt{TRUE}{-}1)$$

$$\frac{M, \Gamma \vdash p_a \hookrightarrow S \qquad M, \Gamma \vdash p_b \hookrightarrow \emptyset}{M, \Gamma \vdash p_a \vee p_b \hookrightarrow \varnothing}\,(\vee{-}\mathtt{TRUE}{-}2)$$

$$\frac{\begin{array}{c}M, \Gamma \vdash p_a \hookrightarrow \{(r_a^1, \mathbf{x}_a^1, m_a^1), \ldots, (r_a^k, \mathbf{x}_a^k, m_a^k)\}\\ M, \Gamma \vdash p_b \hookrightarrow \{(r_b^1, \mathbf{x}_b^1, m_b^1), \ldots, (r_b^j, \mathbf{x}_b^j, m_b^j)\}\\ k \geq 1 \qquad j \geq 1\end{array}}{\begin{array}{c}M, \Gamma \vdash p_a \vee p_b \hookrightarrow \{(\bullet,\\ \mathbf{x}_a^1 \cup \ldots \cup \mathbf{x}_a^k \cup \mathbf{x}_b^1 \cup \ldots \cup \mathbf{x}_b^j,\\ m_a^1 + \text{“ and ”} + \ldots + \text{“ and ”} + m_a^k +\\ \text{“ , or ”} + m_b^1 + \text{“ and ”} + \ldots + \text{“ and ”} + m_b^j)\}\end{array}}\,(\vee{-}\mathtt{FALSE})$$

D. Assigning a responsible object

When generating errors for a failing universal quantifier, we would like to see which quantified objects caused the failure rather than a single failure for the entire predicate. This allows the user to determine which parts of the system failed and to work on each task separately. Therefore, like the conjunction rules, the rules for universal quantification split errors and produce separate sets of errors for each failing object. Additionally, the rules for universal quantification must determine the responsible object for the error by using the heuristic to assign each error to the object bound in the nearest universal quantifier. Therefore, ERL produces the following error messages for the *quantifier* specification in Table II:

> *InputT3 must declare the type resultPort.*
> (Responsible object: InputT3)
> *InputT4 must declare the type resultPort.*
> (Responsible object: InputT4)

The rules for universal quantification are:

$$\frac{M.\mathsf{items}(\Gamma, L) = \emptyset}{M, \Gamma \vdash \forall x \in L\ .\ p \hookrightarrow \emptyset}\,(\forall{-}0)$$

$$\frac{\begin{array}{c}M.\mathsf{items}(\Gamma, L) = \{o_1, \ldots, o_n\} \qquad n \geq 1\\ M, \Gamma[x \mapsto o_1] \vdash p \hookrightarrow S_1\\ \vdots\\ M, \Gamma[x \mapsto o_n] \vdash p \hookrightarrow S_n\\ (x\ fresh\ in\ \Gamma)\end{array}}{\begin{array}{c}M, \Gamma \vdash \forall x \in L\ .\ p \hookrightarrow\\ \{(o_i, \mathbf{x}/x, m) \mid (\bullet, \mathbf{x}, m) \in S_i \wedge x \in \mathbf{x} \wedge i \in \{1 \ldots n\}\} \cup\\ \{(r, \mathbf{x}/x, m) \mid\\ (r, \mathbf{x}, m) \in S_i \wedge (x \notin \mathbf{x} \vee r \neq \bullet) \wedge i \in \{1 \ldots n\}\}\end{array}}\,(\forall{-}N)$$

The rules bind each object individually, generate errors from the inner predicate, and then substitute in the responsible object if the error does not already have one and if the message uses the variable bound by the quantifier. The end result is the heuristic that the responsible object is inserted by the inner-most quantifier of the failing predicate. This makes sense since the inner-most quantifier may rely on outer quantifiers to produce the quantification list. We have also observed that, in general, specification writers order quantifiers with the most general variables first and then proceed into more specific variables. Even if the selected object is not truly the root cause of the failure, the rule does guarantee that the object is used in the error message due to its inclusion in $\mathbf{x}$ and therefore plays some role in the failure.

If there is no universal quantifier that bound a free variable in the error message, it is possible for ERL to return errors which do not have a responsible object. Since there are no universal quantifiers, the variables must either have been introduced by an existential or be pre-defined. In this case, ERL uses the default responsible object that would have been used by the host system. Acme defaults to the object which defined the specification, the `self` object, since this is the only known object.

This rule requires that all quantified lists in the host must be finite. This is true in the systems we expect ERL to work for, as the host system must be able to test each item in the list. [4]

Universal quantifiers make a clear case for when splitting is important. Consider a specification which quantifies over a list of 100 elements, and 10 of these elements cause a failure, possibly failing in different ways. Instead of a single error, ERL will produce 10 errors. Each error would be associated to a distinct object, and the error messages themselves would be different forms if the specification failed in different ways for each variable binding.

E. *Relying on current state*

In most cases, ERL creates error messages in the deontic mode and describes a correction that the user must make to for the specification to be correct. However, ERL must sometimes describe the current state of the system to the user, such as in the rules for implication and existential quantification. In the rules for implication, ERL provides the user with information about how the error was triggered.

For the *other* error in Table II, ERL produces:

> *outputT0 must declare the type outputPort since outputT0 declares the type compilePort.*
> (Responsible object: outputT0)

ERL produced this by asking the oracle for the text on the left side of the implication stated as a fact rather than as an instruction, as described earlier in Figure 3. The rules for implication are:

[4]If the list was infinite, then the host system must have theorem-proving capabilities in order to test the predicate. ERL is not expected to work under these environments without significant modification.

$$\frac{M,\Gamma \vdash p_a \hookrightarrow \emptyset \qquad M,\Gamma \vdash p_b \hookrightarrow \emptyset}{M,\Gamma \vdash p_a \implies p_b \hookrightarrow \emptyset}(\implies-\texttt{TRUE1})$$

$$\frac{M,\Gamma \vdash p_a \hookrightarrow S_a \qquad M,\Gamma \vdash p_b \hookrightarrow S_b \qquad S_a \neq \emptyset}{M,\Gamma \vdash p_a \implies p_b \hookrightarrow \emptyset}(\implies-\texttt{TRUE2})$$

$$\frac{M,\Gamma \vdash p_a \hookrightarrow \emptyset \qquad M,\Gamma \vdash p_b \hookrightarrow S \qquad S \neq \emptyset}{\begin{array}{c}M,\Gamma \vdash p_a \implies p_b \hookrightarrow \{(r, \mathbf{x} \cup M.\mathsf{freevars}(p_a)),\\ m + \text{“ since ”} + M.\mathsf{text}(\Gamma, p_a, \texttt{false}, \texttt{false})) \mid \\ (r,\mathbf{x},m) \in S\}\end{array}}(\implies-\texttt{FALSE})$$

The rules for the existential quantifier, below, also take advantage of this message form. Like disjunction, exists must join the current error sets. While this results in a relatively longer message, it contains only the keywords that the user needs.

$$\frac{\begin{array}{c}M.\mathsf{items}(\Gamma, L) = \{o_1, \ldots, o_n\}\\ M,\Gamma[x \mapsto o_1] \vdash p \hookrightarrow S_1 \ \ldots\ M,\Gamma[x \mapsto o_n] \vdash p \hookrightarrow S_n\\ S_1 = \emptyset \vee \ldots \vee S_n = \emptyset\\ n \geq 1 \qquad (x\ fresh\ in\ \Gamma)\end{array}}{M,\Gamma \vdash \exists x \in L\ .\ p \hookrightarrow \emptyset}(\exists-\texttt{TRUE})$$

$$\frac{M.\mathsf{items}(\Gamma, L) = \emptyset \qquad (x\ fresh\ in\ \Gamma)}{\begin{array}{c}M,\Gamma \vdash \exists x \in L\ .\ p \hookrightarrow \quad \{(\bullet, \text{“There exists no ”} + x\\ + \text{“ such that ”} + M.\mathsf{text}(\Gamma, p, \texttt{false}, \texttt{false}))\}\end{array}}(\exists-\texttt{FALSE}-0)$$

$$\frac{\begin{array}{c}M.\mathsf{items}(\Gamma, L) = \{o_1, \ldots, o_n\} \qquad n \geq 1\\ M,\Gamma[x \mapsto o_1] \vdash p \hookrightarrow S_1\\ \vdots\\ M,\Gamma[x \mapsto o_n] \vdash p \hookrightarrow S_n\\ S_1 \neq \emptyset \wedge \ldots \wedge S_n \neq \emptyset \qquad (x\ fresh\ in\ \Gamma)\end{array}}{\begin{array}{c}M,\Gamma \vdash \exists x \in L\ .\ p \hookrightarrow\\ \{(\bullet, M.\mathsf{freevars}(p), \text{“There exists no ”} + x\\ + \text{“ such that ”} + M.\mathsf{text}(\Gamma, p, \texttt{false}, \texttt{false}))\}\end{array}}(\exists-\texttt{FALSE}-N)$$

F. *Negation*

ERL handles negation predicates separately from the other predicates. If simply we print out “not”, or an equivalent negative, anytime we see the predicate, we can introduce ambiguity and double (or more!) negatives. Instead, ERL first normalizes the specification by pushing not predicates inward to atomic predicates, and then it requests that the oracle provide a reasonable negation messages for atomic predicates. The oracle then produces single negative phrases, examples of which are shown earlier in Figure 3. As most atomic messages are a single phrase, we have pushed the negatives down to a level where they are unambiguous and understandable. The rules for negation below assume that the specification has already been normalized.

$$\frac{M.\mathsf{evaluate}(\Gamma, a) = \texttt{false}}{M,\Gamma \vdash \neg a \hookrightarrow \emptyset}(\neg-\texttt{TRUE})$$

$$\frac{M.\mathsf{evaluate}(\Gamma, a) = \texttt{true}}{\begin{array}{c}M,\Gamma \vdash \neg a \hookrightarrow\\ \{(\bullet, M.\mathsf{freevars}(a), M.\mathsf{text}(\Gamma, a, \texttt{true}, \texttt{true})\}\end{array}}(\neg-\texttt{FALSE})$$

IV. Implementation of ERL

We implemented the ERL rules in Prolog, and we provided a Java wrapper and interface for the oracle. For a system to use ERL, it must be able to transform its specifications into the types defined by ERL, and it must provide an implementation of the oracle.

We implemented a transformer and oracle for AcmeStudio. The ERL addition to Acme required 139 LOC for the transforming functionality, and 643 LOC for the oracle. Of the lines of code for the oracle, 486 LOC were for generating messages for atomic predicates and retrieving the names of elements in Γ. The remaining code was used to interact with AcmeStudio's existing typechecker. Acme utilizes all of the rules described in Section III.

V. User study results

As discussed in Section II, we ran a small user study where each participant attempted to fix errors in two Acme architectures. The users were provided with ERL for one of the two architectures. Both architectures contained five failing specifications, as described in Table II, and ERL expanded these into seven distinct errors due to splitting from conjunctions and universal quantifiers. The qualitative data suggests that ERL is helpful for many users, particularly for complex specifications. When it was not helpful, it did not misdirect or otherwise hinder users.

A. Results by type of error

For the *simple* errors, as defined in Table II, users did not receive any additional benefit from ERL. The graphical indicators were already in the correct place in the control configuration, and the specification was short enough that users could quickly find the problem. Additionally, the rule names themselves were descriptive enough that the errors were fairly obvious. Users almost always went directly to the cause and guessed what the problem was without reading the error message, so ERL did not help or hinder in this case.

One problem we noted was that fully qualified names in error messages confused participants. Upon seeing a qualified name, participants were overwhelmed by the number of words, so we have removed this from ERL. The graphical indicators already point the user to the location of the objects, so there should be little information lost. The participants in this study did receive error messages with fully qualified names, and we expect that this change would have improved the overall results.

As expected, ERL was much more helpful for *conjunction* errors. Participant D, who made several comments about not knowing which side of a conjunction was failing during the control portion of the study, was clearly helped by the ERL error messages. When using ERL, this same participant read the error message for the conjunction failures and fixed both errors in approximately three minutes.

The results of the *disjunction* error were surprisingly mixed. While we expected the wordiness to bother participants, participants A and D strongly preferred the ERL error message to using the system without ERL. When participant D initially opened the second system, he expressed concern that the errors were going to be as difficult to fix as before:

> "Ugh, it's all typecheck [errors] still..."

After examining a few error messages, the participant chose to start with the disjunction error and fixed it within a few minutes by doing what the error message suggested. Upon seeing the error go away, the participant commented:

> "So, this seems like not too much thought."

Participant B found that the error message was "not at all helpful", though participant B did not find any of the error messages helpful.

ERL appeared to help participants A, C, and D when fixing errors that came from failing *existentials* and failing *implications*. In particular, ERL helped clear up confusion about variable bindings. In the control part of the study, Participant A was slightly confused by the implication specification in Table II. The participant believed that two *different* ports had to be a compile port and an output port. Participant A read the specification and examined the seemingly correct system several times before finally realizing the confusion. When participant C encountered this error with the ERL message instead, the participant did not even have the opportunity to be confused. The ERL error message replaced the variable `p` with the specific port name `outputT0`, and the participant clearly understood that this port had to be both an output port and a compile port. The only time participants saw variables in the error messages was when they encountered a failure from an existential. Participant D did not appear to be bothered by this, while participant A would jump to the source to understand the error better.

The true test of ERL was the *universal quantification* errors. These errors were generally the hardest to fix as they were the most complex, they were declared at the system level, and they failed in two places in the architecture. For these errors, ERL was clearly an improvement over the control system. In the control, participants narrowed down their search by reading the specification, but they still had problems after that. In the Build example, there were four objects that were being quantified over, and participants had to carefully inspect each one. They discovered the problem by carefully exploring each of the four objects and noticing that two were slightly different. Then they went back to the specification, determined which set of two objects were causing the problems, and corrected them. However, participant C believed that the correct connectors were the incorrect ones, and accidentally "fixed" the wrong connectors! The participant realized the mistake after the tool did not remove any errors when rechecking the system. The ERL errors were clearly helpful in these cases, and participants appeared less frustrated during their search for the root cause.

B. Participant impressions of the ERL messages

After the users fixed the errors in both architectures, they took a post-survey about the error messages that they saw. Both participants C and D preferred the ERL messages. Participant

A believed the two configurations were very similar and noticed little difference between the error messages. Interestingly, this participant used and was clearly helped by the error messages during the study. The error messages were possibly unobtrusive enough that the difference did not register to the user given all the other features of AcmeStudio.

Participant B preferred to just know which specification failed and view the source directly. However, the participant chose not to read the specification source when using ERL, even though the source was equally available in both systems. This participant switched quickly between tasks in both parts of the study and did not appear to spend much time focusing on the errors. The participant also completed very few tasks during the study and had to be stopped due to time constraints.

From this qualitative data, we believe that the error messages provided by ERL certainly help with some kinds of failures, and some users clearly prefer them. In *no* situations did ERL misinform the users, lead them away from the cause of the error, or otherwise hinder their progress. In each situation, it either helped or had no effect on their progress towards finding the error, other than a few seconds to read the message. For this reason, ERL is being put into use within AcmeStudio.

VI. Complex Examples from MDS

The Acme specifications in the case study were created for the purpose of the study, so in this section we explore how ERL handles a real Acme specification. For this purpose, we will use the Mission Data System (MDS), one of the most complex architectures specified in Acme. MDS specifies a state-based reactive control architecture for space systems. More about MDS and its Acme specification can be found in [6].

What makes MDS so complex is the number of constraints between two or more architectural elements. In order to express these constraints in Acme, the user needs a universal quantifier for each element, plus quantifiers over the sub-elements that attach larger elements together. The end result is that in order to specify a constraint between n elements, we may need $2n$ quantifications. For this reason, it is not uncommon for Acme specifications to have four or more quantifiers.

Given the complexity of these specifications, the writers of MDS also added generic error messages to each specification. In this section, we will compare these generic error messages to the specific error messages provided by ERL.

Figure 4 is a sample of a specification, and all necessary sub-specifications, from MDS. As we can see from the error message, this rule checks that only estimators receive commands from actuators. The specification `rule112` calls out to a sub-specification to do the work. If this specification fails because an Estimator's port was properly connected, but not of type CommandNotifProvPortT, the original version of AcmeStudio would give the error message:

> *Rule 2.3: An Actuator may only notify estimators of commands*

Since the rule is defined at the system level, the user would have to investigate every connection between actuators and other components. To make matters more confusing, the user would probably look for an Actuator that is connected to something that is not an Estimator, when the real problem is the port type of the Estimator's port.

ERL would have produced the error message:[5]

> *estPort must declare the type CommandNotifProvPortT since actPort is connected to estPort.*

and would direct the user to estPort, the port on the Estimator which is causing the failure. If this failure occurred multiple places in the system, then ERL would produce a distinct error for each failing port.

Another MDS rule checks that a component does not connect twice to a port on another component. Like the previous specification, this specification has several quantifiers and eventually has an implication that checks whether some ports are connected incorrect. The generic error message for this rule is:

> *Rule 10: No two ports of a component should be connected to the same target port.*

This does explain the problem that the specification is trying to find, but it doesn't tell us which ports are the problem. If a component had two ports, `portA` and `portB` that both eventually connect to `otherPort`, then ERL would produce the specific error message:

> *portA must equal portB since portA is connected to otherPort and portB is connected to otherPort.*

Ideally, the user should see both the generic and specific messages. The generic description provides the user with the specification intent and would help the user understand the system goals. However, the ERL error message provides actionable guidance for how to fix the current error.

The last MDS example we consider checks that Sensors are in the correct state based upon how many Estimators are listening for data. The generic message for this rule is:

> *Rule 4.4: A sensor that it not connected to any estimators should specify that it is only raw data; if it is connected to more than one estimator, it should specify that it is informative to more than one.*

Of course, only one of these two things could be true at any point; the sensor can not be hooked up to no estimators and more than one estimator at once. With ERL, not only does the user find out which sensor is causing the problem, they also find out which predicate is actually breaking and receive direct guidance on how to fix the error:

> *mySensor.rawData must equal true since the number of estimators connected to mySensorPort is 0.*

While the generic messages do help us understand the purpose of the specification and prevent us from making future errors, they do not help a user find the cause of their error. This is particularly important for specifications as complicated

[5]The ability to descend into an Acme sub-specification is currently being implemented.

```
rule rule112 = R2_3(self)
 <<label : string = ''Rule 2.3: An Actuator may only notify estimators of commands''; >> ;

analysis R2_3(sys : system) : boolean =
 (forall compA : ActuatorT in sys.COMPONENTS |
  forall pA : CommandNotifReqrPortT in compA.PORTS |
   forall compX : Component in sys.COMPONENTS |
    forall pX : Port in compX.PORTS | connected(pA, pX) ->
     (declaresType(compX, EstimatorT) and declaresType(pX, CommandNotifProvPortT)));
```

Fig. 4. MDS Specification

as MDS; even if the control system using MDS is small, it is still difficult to parse through the specifications by hand. If the system itself is also large, the user must spend a great deal of time checking parts of the system that are already correct. The user study and MDS examples show that ERL error messages are a useful addition to the existing error reporting mechanisms because they help users to find the root cause of the error, even in complex specifications and large systems.

VII. Related Work

Shapiro [5] explored and formalized algorithms for how programmers debug logic programs. Shapiro's algorithm for debugging a system with incorrect output is the most similar to the algorithm we have proposed. Like Shapiro, we investigate the sub-predicates for the source of the error, and we use an independent oracle to determine the correctness of a sub predicate. However, Shapiro's algorithm stops at the first failing sub predicate. Our algorithm continues to gather all of the failure points in the predicate, as well as produces them into a human readable error message. Additionally, ERL uses a heuristic to identify a responsible object for the error so that the user receives direction on the failing object, not just the failing specification.

ESC/Java uses an error reporting mechanism that also aims to provide the user with a failure point and a directed error message [7]. However, the error reporting mechanism is inherently different from ERL because ESC/Java's checks that the specifications hold true universally as a set, while tools such as Acme check that individual specifications hold true. Since ESC/Java's specifications must hold true together, the theorem prover can not break apart the specifications and check them individually the way ERL's oracle does. It is the oracle's ability to analyze sub-predicates of the specification that allow ERL to find the root cause of the error and provide the directed message. For ESC/Java, [7] can not find the root cause of the error, but it does display the point where the theorem prover found a counter-example to its proof. To show the user how it got into this bad state, the ESC/Java error generator creates a trace based upon labels it leaves in the logical predicates.

ESC/Java also has slightly different goals from ERL due to the way their users fix errors. The work on ESC/Java attempts to generate fewer errors and condense them; ESC/Java produces one error for each method rather than one error for each failing path. ERL attempts to do the opposite; it splits the errors at every opportunity. This difference makes sense when we consider how the users find and fix these errors. A user of ESC/Java works on the entire method and considers the whole problem as a single error. On the other hand, a user of Acme regards multiple failures from a universal quantifier as different errors. While the errors were all generated by the same specification, they are about different parts of the system and likely are not related.

The model checking community has also investigated error reporting [8]; the work which is closest to ours is that of [9]. The goals stated in [9] are very similar to the ones we present; they look to get at the cause of an error trace from a model checker, rather than the symptom. When they determine the cause of the error, they then produce one error trace for each cause and generate separate error traces for each cause. The main hindrance is that, like the work with ESC/Java, it is difficult to treat a model checker as an oracle because it can not analyze sub predicates individually. Ball et al. proposed a heuristic for the problem by using correct traces to narrow down the problems in the failing traces. This heuristic allows for a model checker to be treated in a fashion similar to our oracle, but it requires that enough correct traces exist to guide it.

There is a large body of work on messages for typing errors (summarized in [10]). The research which uses program slicing [11] to find the causes of type errors [12]–[14] is the closest to ERL. Program slicing is a technique for analyzing which parts of a program are involved in computing the value of a variable at a particular program point. By analogy, ERL can be viewed as an approach for analyzing which parts of a specification and a model result are responsible for causing the specification to fail on that model. Rather than following data- and control-dependencies in a program, our approach analyzes how the truth of a logical specification depends on the truth of its parts.

Another system for describing typing errors, Seminal [15], uses a similar mechanism as ERL for separating the error-generation system from the checker itself. Seminal also treats the checker as an oracle of knowledge and will break down expressions into sub-expressions in order to find the root cause of a typing error. Upon finding the root cause, Seminal searches for similar sub-expressions that will typecheck, and it suggests the "best" similar sub-expression to the user as an alternative. However, the sub-expression produced by Seminal may not be the sub-expression the user actually wants, and may then mislead the user. While ERL does not currently provide a correct alternative, it also does not provide the user with misleading information. As the two systems provide different kinds of information, we expect that using both techniques would be beneficial for users.

VIII. Conclusion

We have presented error reporting logic (ERL), a system for automatically generating error messages from first-order predicate logic. ERL presents a user with a precise error message by automatically analyzing the specification to select only the predicates involved in the failure. Additionally, it uses a heuristic to assign fault to a particular object so that the user is directed to the point of failure.

Our user study shows that most users were helped by the ERL error messages, particularly in errors from conjunction, disjunction, and universal quantification predicates. ERL provided users with an indication of the source of the error and specific instructions about how to fix the error. When ERL did not help, it also did not mislead the users. This is a large improvement over the control system which did not provide the users with any specific guidance. While general guidance is useful for preventing future problems and providing knowledge for the user, it does not help the user fix the current problem.

The user study also provided some interesting insights into how users find the root cause of the error. In particular, we found that users frequently scan any text for keywords that will lead them to the cause of the error, and they only read text for content if they are stuck or want to confirm their suspicions. During the study, participants also vocalized concern about not knowing which parts of the specification was failing. Finally, we found that participants fell back to pattern recognition when they could not be helped through other mechanisms. ERL addressed all the issues we saw except providing a “good” pattern to follow.

While ERL is certainly useful for Acme and similar specification systems, we anticipate that it will have greater benefit in more complex specification systems. Systems which require more complex logical connectives can easily extend the ERL concepts of splitting and joining errors to produce more useful error messages. ERL may also prove beneficial for systems where specifications are globally distributed by pinpointing only the relevant parts of the global specification. We look forward to seeing how other specification systems may be able to extend the concepts presented in ERL.

Acknowledgements

The authors would like to thank the AcmeStudio team for their feedback and assistance, especially Bradley Schmerl and Nicholas Sherman. This work was supported in part by NSF grant CCF-0546550, DARPA contract HR00110710019, the Department of Defense, Army Research Office grant number DAAD19-02-1-0389 entitled Perpetually Available and Secure Information Systems, the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation, and a fellowship from Los Alamos National Laboratory.

References

[1] D. Garlan, R. Monroe, and D. Wile, “Acme: an architecture description interchange language,” in *Conf. of the Centre for Advanced Studies on Collaborative research*, 1997.

[2] D. Hou and H. Hoover, “Using SCL to specify and check design intent in source code,” *Trans. on Software Eng.*, 2006.

[3] D. Jackson, “Alloy: a lightweight object modelling notation,” *Trans. Softw. Eng. Methodol.*, 2002.

[4] “AcmeStudio,” http://www.cs.cmu.edu/ acme/AcmeStudio/.

[5] E. Y. Shapiro, “Algorithmic program diagnosis,” in *9th Principles of programming languages*, 1982.

[6] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, “Software architecture themes in JPL’s Mission Data System,” *IEEE Aerospace Conf. Proc.*, vol. 7, pp. 259–268 vol.7, 2000.

[7] K. R. M. Leino, T. Millstein, and J. B. Saxe, “Generating error traces from verification-condition counterexamples,” *Sci. Comput. Program.*, 2005.

[8] A. Groce and W. Visser, “What went wrong: Explaining counterexamples,” in *10th Intl. SPIN Workshop*, 2003.

[9] T. Ball, M. Naik, and S. K. Rajamani, “From symptom to cause: localizing errors in counterexample traces,” in *Principles of programming languages*, 2003.

[10] B. Heeren, “Top quality type error messages,” PhD thesis, Universiteit Utrecht, The Netherlands, 2005.

[11] M. Weiser, “Program slicing,” *Trans. Software Engineering*, July 1984.

[12] V. Choppella and C. Haynes, “Diagnosis of ill-typed programs,” Indiana University, Tech. Rep. 426, 1994.

[13] C. Haack and J. B. Wells, “Type error slicing in implicitly typed higher-order languages,” *Sci. Comput. Program.*, vol. 50, no. 1-3, pp. 189–224, 2004.

[14] F. Tip and T. B. Dinesh, “A slicing-based approach for locating type errors,” *Trans. Sfw. Eng. Methd.*, vol. 10, no. 1, pp. 5–55, 2001.

[15] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, “Searching for type-error messages,” in *Programming language design and implementation*, 2007, pp. 425–434.